Implementing Bags on a Shared Memory MIMD-Machine

Herbert Kuchen Katia Gladitz
RWTH Aachen, Lehrstuhl für Informatik II*

Abstract

Multisets (also called bags) are an interesting data structure for parallely implemented functional programming languages, since they do not force an unneeded restriction of the data flow and allow to exploit as much parallelism as possible. Most operations on multisets can be understood as special cases of the so-called Gamma scheme [BL90]. In the present paper, we investigate efficient implementations of several instances of this Gamma scheme on MIMD-machines with shared memory.

1 Introduction

The ubiquitous data structure in functional (and logic) programming language is the list. For a parallel implementation of such a language, however, lists have the drawback that they force a sequential access to their elements. Hence, it is only interesting to use lists, if the computation related to every element is large compared to the length of the list. Several researchers have investigated other applicative data structures which are more adequate for parallel implementations, among them modifications of classical array implementations, e.g. I-structures [AN90] and distributed applicative arrays [KG90], as well as tree structured implementations of array-like data structures, e.g AVL-dags [My84], quad trees [Wi87], sequences [LKID89, Ku90, KG90]. "Array-like" means that an element can be "efficiently" accessed using its index. This property allows to access each element independently of the others and makes arrays suited for parallel computations.

The tree structured implementations were originally designed to enable a storage saving, overlapping representation of different "versions" of a data structure. These versions occur due to the side-effect-free nature of functional languages. A data structure must not be modified, but it is only allowed to generate a new version of it, which differs from the original version in certain parts. Tree structures have the disadvantage that all the accesses to the elements (usually stored in the leaves) have to pass the root, which is a bottle-neck and may sequentialize the accesses (at least on distributed memory machines).

Another data structure, which is especially interesting for parallely implemented functional languages, is the *multiset* or *bag*, i.e. a variant of sets, where multiple occurrences of each element are allowed. For a parallel implementation, it offers even more flexibility than an array, since the elements are unordered and arbitrary elements may be combined.

It is good functional programming style, not to access a data structure element by element but to use some dedicated higher order functions, e.g. appropriate variants of the ubiquitous

*Ahornstr. 55, W-5100 Aachen, Germany, {herbert,katia}@zeus.informatik.rwth-aachen.de

functions map and fold, which treat the data structure as a whole (monolithically). This has the advantage of a higher level of programming and leads to efficient (parallel) programs, provided that these higher order functions have been efficiently implemented (in parallel). Moreover, less versions of a data structure need to be build than element by element modifications would require.

Map and fold can be easily carried over to bags:

$$\begin{aligned} \text{map f } \{b_1, \ldots, b_n\} &:= \{f\ b_1, \ldots, f\ b_n\} \\ \text{fold } (\oplus)\ \{b_1, \ldots, b_n\} &:= b_1 \oplus b_2 \oplus \ldots \oplus b_n \end{aligned}$$

where $\oplus$ is an associative and commutative binary function (used in infix notation).

We use the notation $\{\ldots\}$ for a multiset, and the notations $B_1 \cup B_2$ and $B_1 - B_2$ for the union and difference of multisets, respectively. $\#(b, B)$ denotes the number of occurrences of an element b in the multiset B. Then, $\#(b, B_1 \cup B_2) := \#(b, B_1) + \#(b, B_2)$ and $\#(b, B_1 - B_2) := \#(b, B_1) \div \#(b, B_2)$, where $x \div y$ is $x - y$, if $x \geq y$, and 0 otherwise. $b \in B$ is a shorthand for $\#(b, B) > 0$.

Bags are also interesting for parallel imperative languages like Gamma [BL90]. Gamma implements the idea of a chemical reaction of the elements in a multiset. Some elements which fulfill a certain predicate (also called reaction condition) may be taken from the multiset and replaced by new elements which are generated by combining the selected elements using the so-called reaction function. The basis of Gamma is the following scheme (here a bit simplified):

$$\begin{array}{lll} \Gamma\ p\ f\ B := & \text{if} & (\forall b_1, \ldots, b_k \in B)\ \neg(p\ b_1 \ldots b_k) \\ & \text{then} & B \\ & \text{else} & \text{let } b_1, \ldots, b_k \in B \text{ such that } p\ b_1 \ldots b_k \text{ in} \\ & & \Gamma\ p\ f\ (B - \{b_1, \ldots, b_k\} \cup f\ b_1 \ldots b_k) \end{array}$$

A more convenient syntax for this is:

$$\begin{array}{l} \text{replace } b_1, \ldots, b_k : B \\ \text{if } p\ b_1 \ldots b_k \\ \text{by } b_1', \ldots, b_l' : B \end{array}$$

where $\{b_1', \ldots, b_l'\} := f\ b_1 \ldots b_k$, $k \geq 1$, $l \geq 0$. We call this a (k, l, p)-*scheme*. This can easily be generalized to work on several multisets simultanously:

$$\begin{array}{l} \text{replace } b_1 : B_1, \ldots, b_k : B_k \\ \text{if } p\ b_1 \ldots b_k \\ \text{by } b_1' : B_1', \ldots, b_l' : B_l' \end{array}$$

where $B_1, \ldots, B_k, B_1', \ldots, B_l'$ need not be pairwise distinct.

The advantage of the Gamma scheme is that, in principle, each tuple $(b_1, ..., b_k)$ of elements can be handled simultanously. Implementations of Gamma on a MIMD-machine with distributed memory, namely the Intel iPSC2 with 64 processors [BCL88], and on the Connection Machine [Cr91], i.e. a SIMD-machine, have been developed. [Mu91] shows that multisets and the Gamma scheme can be used to elegantly program a broad range of applications. Consequently, it is interesting to include the Gamma scheme as a predefined higher order function into a functional language [Ha91]. Of course, the scheme has to be restricted in such a way that an application of the Gamma scheme leads to one unique result, regardless of the non-determinism during computation. This is required by the referential transparency of functional languages, i.e. the same expression has the same value in the whole program. Restrictions are also needed, since many instances of the Gamma scheme cannot be implemented efficiently. Only such instances are interesting, where elements $b_1, \dots, b_k \in B$ fulfilling $(p\ b_1 \dots b_k)$ can be selected "directly", and it can be avoided to check $(p\ b_1 \dots b_k)$ for all $b_1, \dots, b_k \in B$.

map and fold on multisets can be seen as (more precisely: can be defined using) special cases of the (general) Gamma scheme, where the reaction condition is always true:

$$\begin{array}{ll} \text{map } f\ B := & B' := \emptyset; \\ & \text{do replace } b : B \text{ if } \mathit{true} \text{ by } (f\ b) : B' \text{ od}; \\ & \text{return } B'. \\ \text{fold } (\oplus)\ B := & \text{do replace } b_1, b_2 : B \text{ if } \mathit{true} \text{ by } (b_1 \oplus b_2) : B \text{ od} \\ & \text{return (oneof B).} \end{array}$$

where (oneof B) returns an arbitrary element of the bag B. Actually, fold is a $(2, 1, \mathit{true}_2)$-scheme, where true_k is a k-ary function which always delivers true.

The rest of the paper is organized as follows. In Section 2, we discuss implementations of map and fold on a shared memory MIMD-machine. In Section 3, we will generalize these special cases in such a way that the resulting schemes are still efficiently implementable. Section 4 contains some experimental results. In Section 5, we conclude and point out future work.

2 Fold and Map

If we assume that the creation of the bag elements takes the same time t for every element and that the computation of $b_1 \oplus b_2$ takes the same time t for all elements b_1 and b_2 than an implementation of fold for bags is straightforward. The elements of the bag are assigned to the leafs of a (balanced) binary tree, and if the two sons b_i and b_j of a node are ready, $b_i \oplus b_j$ is computed giving the bag element $b_{i,j}$, which is assigned to the father node. Finally, the result can found at the root of the tree (see Fig. 1).

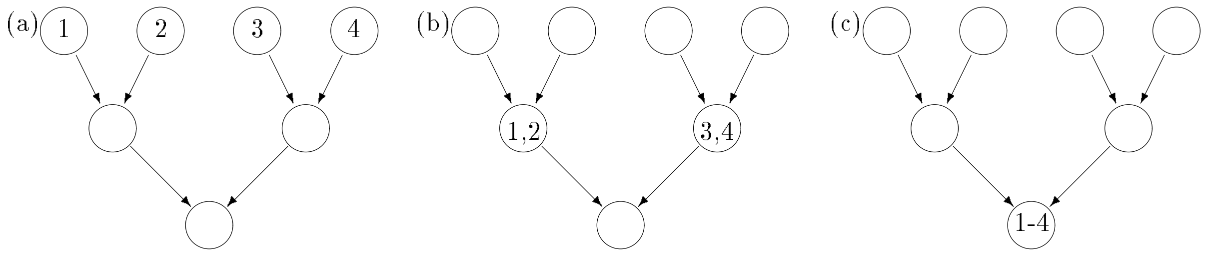

Figure 1: Simple folding scheme.

This approach needs time $t \cdot \log\ n$ using n processors. Unfortunately, the simple approach gives away possible parallelism, if the creation of bag elements does not take the same time for all elements. In some applications, the bag elements have a *size*, which influences the time needed to combine it with another element. The produced bag element often inherits the size from its ancestors. Suppose that in a data base, all objects have to be collected and sorted, which fulfill a certain condition. Each bag element is assigned to a part of the data base and contains the corresponding sorted list of objects, fulfilling the condition. Suppose that $m \gg n$ objects are contained in one bag element and that the other bag elements contain only one object. In order to get a sorted result, the operation $\oplus$ has to merge its operands. To merge two list with n_1 and n_2 elements respectively needs $O(n_1 + n_2)$ steps. In our example, this means that the whole program needs time $O((m+n) \cdot \log n)$ using n processors. A more flexible solution (see below) only needs time $O(m+n)$.

2.1 A More Flexible Algorithm for Fold

The problem with the above simple approach is that it is completely fixed, how the elements have to be connected. A more sophisticated approach could use the associativity and commutativity of $\oplus$ to get more parallelism: if two elements b_1 and b_2 are ready, they are combined by $\oplus$ and deleted. The result $b_1 \oplus b_2$ is inserted into the bag. If a bag element inherits its size from its ancestors as in the above example, the resulting algorithm will not only exploit more parallelism but also minimize the overall run time.

The problem is to detect which elements are ready and can be combined. An ad hoc solution is to use a global storage cell to store the first ready element. If the next element is ready, it finds its partner there, and they are combined. After this, the cell is free again. The disadvantage of this approach is that the global storage cell is a bottle-neck which forces a sequentialization.

The following approach combines the first ready elements without forcing a sequentialization. Below, we show a pseudo-code formulation of the algorithm. The corresponding C-code can be found in Appendix A.1.

Fold Algorithm:

```
Initially: each bag element is assigned to a leaf of a binary tree, and it has weight 1
           each node has no owner
For each bag element b, there is a corresponding process P_b executing the following code:
 current node v is the leaf assigned to b
 if weight(b)= n then return the result b and stop
 while v is not the root
  new current node v is the father of v
  lock v
  if v has no owner
  then take possession of v and unlock it
  else lock owner b' of v
       if b' has already been combined
       then take possession of v, unlock b' and v
       else lock b
            if b has already been combined then unlock b, b' and v, and stop
            release v and unlock b', b and v
            create a new process P_b'' overwriting (the leaf v' corresponding to) b
               by b'' := b ⊕ b'; weight(b'') := weight(b)+weight(b'),
               the initial current node of P_b'' is v'
            stop
```

Each bag element b is computed by an own process P_b and assigned to an (unique) leaf of a (balanced) binary tree. If an element b_1 is ready, P_{b_1} moves to the father v of its current node and locks v. If v has no owner (e.g. initially), P_{b_1} takes possession of v, unlocks v, and moves analogously to the father of v (if v is not yet the root). If v belongs to P_{b_2}, b_2 is a possible partner and it is attempted to combine both elements. In order to do this, (the leaves of) b_1 and b_2 have to be locked (in this order). This ensures that none of these elements is simultanously combined with another bag element. If b_1 was already combined with another element in the meantime, P_{b_1} can be stopped. If b_2 was already combined, we can proceed as if v had no owner.

If b_1 and b_2 are still available, they are combined, and P_{b_1} stops. Nodes belonging to already combined bag elements are handled like nodes without owner, i.e. it is not necessary to change the corresponding owner entries explicitly. The new element $b_1 \oplus b_2$ is assigned to the leaf previously corresponding to b_1.

In order to detect the termination of the fold algorithm, a weight (initially one) is assigned to each of the n bag elements. If two elements are combined, the new element inherits the sum of their weights. If an element has weight n, it is the final result and the algorithm terminates.

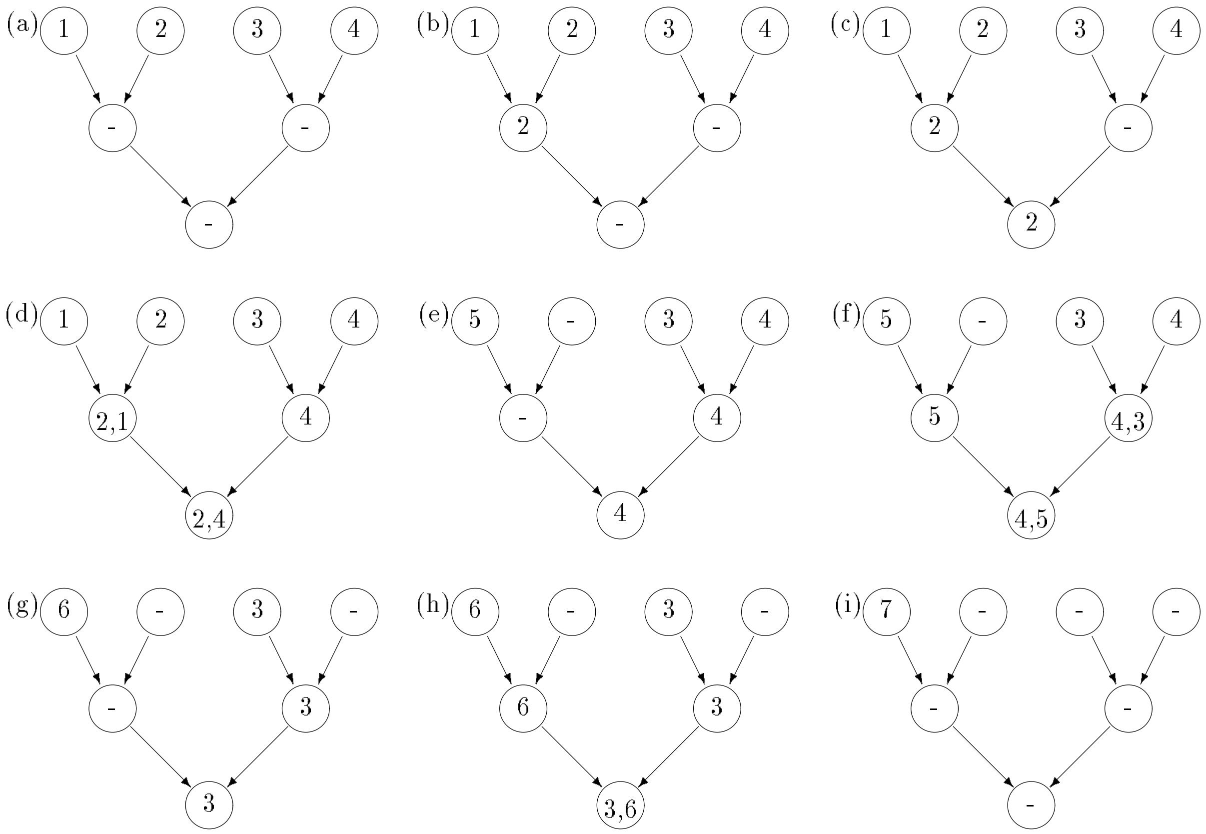

Figure 2: Folding a bag. If a node contains two numbers, the left one indicates the owner and the right one the process which has locked the node.

Fig. 2 shows an example computation. First, P_{b_2} is ready and walks in the direction of the root, taking possession of all encountered nodes (Fig. 2 (a)-(c)). Then, P_{b_1} and P_{b_4} are ready and find nodes belonging to P_{b_2}. They try to lock b_2, but only P_{b_1} succeeds. P_{b_1} locks b_1 and creates a new element $b_5 := b_1 \oplus b_2$ which inherits the leaf of b_1. b_1 and b_2 are unlocked. Now, P_{b_4} succeeds to lock b_2 but notices that it has already been combined. Hence, P_{b_4} takes over the current node (here: the root of the tree) (Fig. 2 (d)-(e)).

Next, P_{b_5} and P_{b_3} are ready and find nodes belonging to P_{b_4}. This time, P_{b_5} succeeds to lock b_4 and creates $b_6 := b_5 \oplus b_4$. P_{b_3} takes over the encountered nodes of P_{b_4} (Fig. 2 (f)-(g)). Finally, b_3 and b_6 are combined to produce the result b_7 (Fig. 2 (h)-(i)). In order to simplify the understanding, references to removed bag elements are not shown in Fig. 2.

Theorem: The fold algorithm is deadlock free.

Proof: First, each process tries to lock the father v of its current node. If this is not possible, another process has locked it. Anyway, the process P_b which has locked v can proceed. If v had no owner, P_b takes possession of it, unlocks v, and proceeds to the father of v. Otherwise, P_b tries to lock the owner $\hat{b}$ of v.

Case (1) $\hat{b}$ is already locked by some process $P_{b'}$:

Case (1.1) $P_{b'}$ can also lock its own bag element b':
$P_{b'}$ can proceed and combine b and b'.

Case (1.2) b' is locked by some other process $P_{b''}$:
the current node v'' of $P_{b''}$ is on the path which $P_{b''}$ took on its way from a leaf to the current node v' of $P_{b'}$. Only nodes on this path are owned by $P_{b'}$. On the other hand, it is clear that v'' belongs to b', since $P_{b''}$ has first locked the owner of the encountered node, namely b'. Hence, $P_{b''}$ is in the same situation than $P_{b'}$: it needs to lock its own bag element after having locked the encountered bag element. But $P_{b''}$ is closer to the leaf where $P_{b'}$ started than $P_{b'}$! Of all processes, which are trying to lock the second (i.e. their own) bag element, at least those processes can continue which are closest to the leaves. Note that there is no competition for leaves.

Case (2) P_b can lock $\hat{b}$:

Case (2.1) P_b can also lock its own bag element P_b:
P_b can proceed and combine b and $\hat{b}$.

Case (2.2) b is already locked by some process $P_{b'}$:
Analogously to Case (1.2), $P_{b'}$ is in the same situation than P_b, namely needing to lock its own bag element, and again the current node of $P_{b'}$ lies on a path from a leaf to the current node v of P_b, i.e. $P_{b'}$ is closer to the leaves than P_b. □

Each non-blocked process P_b may cause up to $2 \cdot \log n - 2$ processes to be blocked: the processes waiting for b and the processes waiting for the owner of the current node of P_b. All members of one of these two sets of processes are currently visiting a node on the same path (of logarithmic length) from a leaf to the root. In practical examples, problems due to blocking rarely occur, since the nodes are only locked for a very short time.

Note that the order of locks is important. If a process would lock its own bag element before the encountered one, a situation could arise where up to $n-1$ processes are blocked and only one process can proceed.

2.2 Another Fold Algorithm

A variant of the above algorithm needs less locks. The corresponding C-code can be found in Appendix A.2. The main difference is that, in the tree, there is always at most one reference to each bag element. Before a process P corresponding to a bag element b takes possession of a node, it locks, releases, and unlocks the son v, which it has visisted before. If v does not belong to P any more, this implies that b has already been combined, and P can stop.

Since there is only one reference to each bag element, the bag elements need not be locked.

It is obvious that this algorithm does not cause deadlocks. Each process locks its current node v and then the son of v which it has visited before. At least the processes which have locked the nodes closest to the leaves can proceed. There is no competition for leaf nodes, since they are only visited by one (living) process.

If there are $m \leq \log n$ processes having locked the nodes on a path from some node v to the root, and if each of these processes tries to lock the son of its current node than only one process can proceed (the process at v) and the rest is blocked. Up to $m+1$ additional processes may be blocked because they want to lock one of these nodes as their current node. That means one process can cause $2 \cdot \log n$ processes to be blocked. Note that only three processes can try to lock the same node simultanously, the process at the father and one process coming from each son. The son nodes still belong to the blocked processes, and, if a later process arrives at one of the son nodes, it will use the bag element corresponding to the owner process to create a new process, and then it will stop.

Again, there is no problem with locking in practical programs, since the nodes are only locked for a very short time.

The modified algorithm has the disadvantage that each process usually has to move closer to the root than before, in order to find a bag element. The reason is that old references to bag elements are deleted, and it is possible that two processes pass the same node without noticing it.

2.3 Map

In order to compute (map f B), all elements $(f\ b_i)$ $(1 \leq i \leq n)$ of the produced bag are computed in parallel. The only problem is to detect, when the operation terminates. If a global counter were used for this purpose, this would sequentialize the synchronization phase. Hence, we use a binary tree of counters protected by semaphores. Each counter is initialized with 2 and decremented if one of its sons is ready. If a counter is zero, the corresponding node is ready. The whole operation is ready, if the root is ready.

Actually, this approach is similar to the simple fold algorithm (depicted in Fig. 1). The synchronization phase needs logarithmic time on $n/2$ processors.

3 Generalizations

Up to now, we have discussed how to handle a $(2, 1, true_2)$-scheme (fold). Now, we want to generalize these approaches step by step.

3.1 The Scheme $(k, l, true_k)$

A $(k, 1, true_k)$-scheme with $k > 2$ can be handled similar to fold. Now, each node may have up to $k-1$ owners. When the k-th process arrives, they are combined. The last fold algorithm is preferable, since the leaves need not be locked. When a process releases a node the other owners remain. The algorithm terminates, if the sum of weights of all the owners of a considered node is n. Note that the final result may consist of up to $k-1$ elements.

A $(k, l, true_2)$-scheme with $k > 2$ and $2 \leq l < k$ can be handled analogously. Instead of one new process, l new processes are created, which inherit l of the k leaves of their ancestors. The sum of weights of the ancestors is divided onto the new processes.

Note that a $(k, l, true_2)$-scheme with $l \geq k$ represents an infinite computation.

3.2 The Scheme $(k, 0, p)$

The scheme $(k, 0, p)$ is handled like a $(k, 1, p)$-scheme, except that the new process, which is created when k bag elements are combined is a dummy process. If a dummy process encounters another process, it gives its weight to this process and terminates. This trick is needed in order to detect termination. If the final process is a dummy process, the result is the empty bag.

3.3 The Scheme $(2, 1, \leq)$ and Similar Schemes

In the sequel, we will assume that the elements of the bag are taken from a totally ordered domain. The nice property of the predicate "$\leq$" is that $\neg(b_1 \leq b_2)$ implies $(b_2 \leq b_1)$. This means that we can proceed like in the $(2, 1, true_2)$-scheme. The only difference is that, if necessary, the arguments have to be exchanged before applying the reaction function.

This idea can be generalized to all predicates p, where $\neg(p\ b_1 \ldots b_k)$ implies $(p\ b_{i_1} \ldots b_{i_k})$ for some permutation of the arguments (i.e. $\{1, \ldots, k\} = \{i_1, \ldots, i_k\}$). This includes e.g. the predicate "$\geq$", but not "$<$" and "$>$".

In order to cope with the $(2, 1, <)$-scheme and the $(2, 1, >)$-scheme, another generalization is needed. If the process P_b corresponding to some element b arrives at a node v, where the first current owner b' of v is equal to b, b and b' are not combined, but b gets an additional owner of v. As before, b and b' are exchanged, if $b' < b$ holds instead of $b < b'$ (analogously for "$>$"). Note that it is sufficient to take into account the first owner b' of v, since all other owners are equal to b'. The algorithm terminates, if the sum of weights of all the owners of a considered node is n. The final result may consist of an arbitrary number of equal elements.

The same idea can be used for a scheme $(2, 1, p)$, where the predicate p only depends on one argument, e.g. $p\ b_1\ b_2\ :=\ b_1 > 0$. Here, two elements are only combined, if one of them fulfills the right hand side of the definition of p. Otherwise, the bag element corresponding to the current process gets an additional owner of the current node, and the process stops.

The $(2, 1, \neq)$-scheme can be handled analogously to the $(2, 1, <)$-scheme, except that the exchanging of elements is omitted.

3.4 The Scheme $(2, l, =)$

"$=$" induces a decomposition of the bag elements into equivalence classes. If the maximal number of bag elements which belong to the same equivalence class is small compared to the number of processors, the following approach can be used.

3.4.1 Small Equivalence Classes

A hash table is used, where each entry can store a bag element (protected by a semaphore). For each bag element b, there is a corresponding process P_b. P_b looks up the hash table (by applying the hash function h to b) in order to find a partner b' with $b = b'$. If it finds such a b', b' is removed from the table, b and b' are combined, and a new process is created for each of the l generated bag elements. Otherwise, if there is no such entry, a new entry for b is inserted into the hash table and P_b stops. This approach is independent of the chosen collision handling strategy.

The size of the hash table should be big enough to avoid too many collisions. Note that $l \geq 2$ is allowed, i.e. the bag may grow. If the maximal size of the bag is not known in advance, a dynamic hashing scheme, like linear hashing [Li80] or extendible hashing [FNPS79], may be used.

A process queue is used to maintain the existing processes. An idle processor takes a process and executes it. The algorithm terminates, if the process queue is empty and all processors are idle. Of course, the process queue has to be implemented in such a way that it does not constitute a bottleneck. See, for example, [He87] for a discussion of possible process queue implementations. One possibility is that each processor has its own queue, and the whole process queue consists of a cyclic connection of all these queues. If a processor cannot find a process in its local queue, it searches the cycle (e.g.) clockwise. New processes are always inserted into the own queue.

If the possible equivalence classes are known in advance, the hash table can be replaced by a table where each entry directly corresponds to one equivalence class.

3.4.2 Large Equivalence Classes

If the maximal size of an equivalence class is large compared to the number π of processors, each hash table entry would constitute a bottle-neck. If the possible equivalence classes are known in advance and their number is small, the following approach can be used, provided that the maximal size n_b of the equivalence class C_b of b is also known for every bag element b. For each equivalence class C, a tree T_C similar to that in the $(2, 1, true_2)$-scheme is used.

A problem is to ensure that no bottlenecks are introduced while inserting bag elements into these trees. This can, for example, be solved in the following way. T_{C_b} has n_b leaves. Each processor i $(1 \leq i \leq \pi)$ maintains a list of free leaves $L_{i,C}$ for every equivalence class C. Initially, $L_{i,C}$ contains e.g. the leaves $i, i+\pi, i+2\pi, \ldots$. In contrast to the $(2, 1, true_2)$-scheme, a new element may have to be inserted into another tree. More precisely, if processor i executes a process, which inserts a new element b into T_C, b is assigned to the a free leaf of T_C.

If a leaf is no longer needed, since the corresponding bag element has been combined, the leaf is inserted into $L_{i,C}$. All lists corresponding to C are cyclically connected, similarly to the process queues. If processor i needs a free leaf in T_C, it first searches in $L_{i,C}$ and then follows the cycle (e.g.) clockwise. This idea for managing free leaves can of course be taken over to other schemes, especially to (k, l, p)-schemes, where $l > k$, or to schemes where l is not a constant but a function depending on the currently considered k bag elements.

Again, the termination is detected, if the process queue is empty and all processors are idle. Weights are not used here.

3.5 Argument Transformations

It is straightforward to generalize the approaches proposed for the predicates "$=$", "$\neq$", "$<$", "$>$", "$\leq$", and "$\geq$" to predicates $(p\ b_1\ b_2)\ :=\ (g\ b_1) \odot (g\ b_2)$, where "$\odot$" is one of the above predicates and g is a unary function. The bag elements only have to be transformed by g before they are compared.

3.6 Conjunctions

Another question is, how to handle conjunctions of the primitive predicates, discussed so far. If the considered predicate p is a conjunction of an equality and another predicate "$\odot$", two cases can be distinguished. If the equivalence classes of "$=$" are sufficiently small, one can first ignore "$\odot$" and use the hash table approach for $(2, 1, =)$. The entries of the hash table are in this case lists of elements $b_1, \ldots, b_r$ such that for all $b, b' \in \{b_1, \ldots, b_r\}$ $(p\ b\ b')$ does not hold. A process corresponding to a bag element b looks up the hash table in order to find a partner. If it could not find a partner in the list of bag elements corresponding to the investigated entry, b is inserted into this list.

The other case is that the equivalence classes of "=" are large. Then, the second approach for the $(2, 1, =)$ scheme is used, but the algorithm for the $(2, 1, \odot)$ scheme is applied instead of the $(2, 1, true_2)$-scheme to handle each equivalence class.

If "$\odot$" is "=", p is an equivalence relation and the approaches for the $(2, 1, =)$-scheme can be used directly.

Conjunctions not involving equations are harder to handle. For a possible solution, refer to Subsection 3.7.

3.7 Other Schemes

For other (k, l, p)-schemes, an approach similar to the $(2, 1, <)$-scheme can be used. A node may have several owners. Suppose that a process P_b corresponding to a bag element b arrives at a node v. If v has less than $k-1$ owners, p cannot be checked yet, and b is added as an additional owner of v. Otherwise, $k-1$ owners $b_1, \ldots, b_{k-1}$ are selected, and p is checked for all permutations of the arguments $b, b_1, \ldots, b_{k-1}$ successively, until one permutation is found such that p evaluates to *true*. Then, the reaction function is applied to the arguments, creating l new processes. If such a permutation could not be found, the next $k-1$ owners are selected, and so on. If no selection of owners and no permutation was successful, b is added as an additional owner of v.

This approach is mainly interesting, if p delivers *true* for "many" argument tuples, i.e. only a few selections and permutations have to be tried in the average case. The properties of the considered predicate p can be used in order to reduce the number of tested selections and permutations. For example, if p is known to be symmetric, only one permutation needs to be tried for the given selection of bag elements.

4 Experimental Results

A prototype implementation has been tested on the shared memory MIMD-Machine Sequent Symmetry S16 with 6 processors, where the processes are not executed by an own processor but distributed onto the available processors. Of course, the presented data structures and algorithms have been designed for "big" machines with hundreds or thousands of processors. Unfortunately, we have no access to such a machine. Consequently, the implementation on the existing machine is misleading, since the drawbacks of the simple approaches do not yet show up. Nevertheless, we present the experimental results for the Sequent in order to give an impression of the overhead of the sophisticated approaches on small machines. The operations on multisets have been implemented in C.

In a first step, we have implemented map and the $(2, 1, true_2)$-scheme, in particular fold. The implementation of the other schemes is in progress.

For comparing the different strategies, we use the following examples: 1) determine the minimum of a given set of integers and 2) mergesort on lists (we merge k lists where $k-1$ lists contain only 1 element and the last list contains k elements). The mergesort example demonstrates the effect of bag elements with different sizes, while the minimum example shows the behaviour for equally sized elements. All input values are generated randomly.

The two examples have been tested with different strategies :

1. the ad-hoc solution (CELL) using a global storage cell to store the first ready element.
2. the simple folding scheme (SFS) (see Figure 1), where only the brothers in the tree can be combined.
3. the flexible fold-Algorithm (FFA) (see Section 2.1 and Appendix A.1).
4. the modified flexible fold-Algorithm (MFA) (see Appendix A.2).

The given times are the average values of 6 experimental program runs. The initialization of the bags is not included in these times.

	Mergesort			Minimum		
number of elements	CELL	SFS	MFA	CELL	SFS	MFA
100	0.162714	0.039756	0.179015	0.014001	0.008448	0.036516
500	4.090844	0.363638	4.344727	0.070640	0.041442	0.219533
1000	18.440431	0.764800	19.311458	0.141374	0.083230	0.472010
5000				0.751711	0.422770	2.810702
10000				1.551553	0.850008	6.023908
20000				3.104055	1.696620	12.751272

Table 1: Run times in seconds on one processor

Table 1 shows the results for only one processor. The simple folding scheme (SFS) is the best one, since it causes the lowest overhead.

number of elements	CELL	SFS	FFA	MFA
100	0.032517	0.028026	0.045813	0.045925
500	0.068105	0.056154	0.098309	0.097656
1000	0.094343	0.071470	0.161432	0.156778
5000	0.313215	0.144770	0.753495	0.674998
10000	0.647097	0.226858	1.600992	1.411385
20000	1.201587	0.392708	3.328811	2.930022

Table 2: Run times for the minimum example in seconds on 6 processors.

On 6 processors, SFS is again the best for the minimum example, since, due to the equally sized bag elements, the overhead of the more sophisticated algorithm (FFA and MFA) does not pay off. CELL is beginning to cause a bottleneck.

Table 3 contains the results for mergesort. Now, FFA and MFA have caught up with SFS. For more "unbalanced" examples, they will be better. CELL is the worst, since the global storage cell now becomes a serious bottleneck.

In Table 4, we compare the runtimes for a different number of processors for strategy MFA. The superlinear speedups are caused by the fact that "big" bag elements are less frequently accessed. For similiar reasons, the runtimes of the individual measurements with the same parameters differ considerably depending on the number of accesses to "big" elements.

number of elements	CELL	SFS	FFA	MFA
100	0.060837	0.056733	0.067914	0.067477
500	0.345536	0.346598	0.354871	0.334375
1000	0.700494	0.697912	0.697742	0.683637
5000	4.228496	4.488384	4.213619	4.287852
10000	11.484751	9.834782	10.455351	9.786295
20000	30.400990	21.563451	22.558900	22.671927

Table 3: Run times for mergesort in seconds on 6 processors.

number of elements	1	2	4	5	6
100	0.179015	0.061704	0.063238	0.066502	0.067477
500	4.344727	0.536192	0.377578	0.361526	0.334375
1000	19.311458	1.494350	0.842240	0.798882	0.683637
5000		26.492707	7.701162	6.244119	4.287852
10000		82.309407	25.834885	19.056816	9.786295
20000			78.017759	53.783963	22.671927

Table 4: Run times in seconds for mergesort depending on the number of processors.

FFA und MFA behave very similar. The examples show that they are only better than SFS, if the sizes of the bag elements are very different. For typical applications of multisets, the simple approach is better, regardless of the fact that it sacrifies some parallelism.

5 Conclusion and Future Work

Bags are an interesting data structure for parallely implemented programming languages, since they do not force an unneeded sequentialization of the computation. Moreover, they can be used to program a broad range of applications. Most operations on bags can be understood as special instances of the Gamma-scheme. We have discussed possible implementations of several instances of the Gamma-scheme, and we have shown that all these considered schemes are efficiently implementable. This has been underlined by experimental results based on a prototype implementation. Most instances of the Gamma-scheme occurring in practical examples can be handled efficiently by one of the presented approaches. Nevertheless, we will continue to investigate implementation techniques for instances of the Gamma-scheme, among them e.g. boolean combinations of primitive predicates.

Many of the presented ideas can be taken over to implementations of bags on MIMD-machines with distributed memory. The investigation of such implementations will also be part of our future work. For distributed memory machines, bags are even more interesting than for shared memory machines, since bags allow to compute as much as possible locally and to avoid communication due to unneeded restrictions of the data flow.

Acknowledgements

We thank John Hughes for helpful discussions and for stimulating us to think about the parallel implementation of bag operations.

References

[AN90] Arvind, R.S. Nikhil: Executing a Program on the MIT Tagged-Token Dataflow Architecture, IEEE Tr. on Computers, Vol. 39(3), 1990, 300-318.

[BCL88] J.-P. Banâtre, A. Coutant, D. Le Métayer: A parallel machine for multiset transformation and its programming style, Future Generation Comput. Systems 4, 1988, 133-144.

[BL90] J.-P. Banâtre, D. Le Métayer: The GAMMA Model and its Discipline of Programming, Science of Programming 15(1), 1990, 55-77.

[Cr91] C. Creveuil: Implementation of Gamma on the Connection Machine, Procs. Workshop on Research Directions in High-Level Parallel Programming Languages, LNCS 574, 1991, 219-230.

[FNPS79] R. Fagin, J. Nievergelt, N. Pippenger, H.R. Strong, Extendible Hashing – A Fast Access Method for Dynamic Files, ACM ToDS 4(3), 1979, 315-344.

[Ha91] R. Harrison: Parallel Programming with Pure Functional Languages, Procs. Workshop on Research Directions in High-Level Parallel Programming Languages, LNCS 574, 1991, 231-252.

[He87] M.V. Hermenegildo: Relating Goal Scheduling, Precedence, and Memory Management in AND-Parallel Execution of Logic Programs, 4th Int. Conf. on Logic Programming, MIT Press, 1987, 556-575.

[Ku90] H. Kuchen: Parallele Implementierung einer funktionalen Programmiersprache auf einem Occam-Transputer-System unter besonderer Berücksichtigung applikativer Datenstrukturen, Thesis, RWTH Aachen, 1990, in German.

[KG90] H. Kuchen, G. Geiler: Distributed Applicative Arrays, Procs. 2nd International Workshop on Impl. of Functional Languages on Parallel Architectures, appeared as: Technical Report no. 90-16, University of Nijmegen, 1990, 223-236.

[Li80] W. Litwin: Linear Hashing: A New Algorithm for Files and Tables Addressing, Int. Conf. on Very Large Data Base Systems, 1980, 260-276.

[LKID89] R. Loogen, H. Kuchen, K. Indermark, W. Damm: Distributed Implementation of Programmed Graph Reduction, Procs. PARLE'89, LNCS 365, 1989, 136-157.

[Mu91] L. Mussat: Parallel Programming with Bags, Procs. Workshop on Research Directions in High-Level Parallel Programming Languages, LNCS 574, 1991, 203-218..

[My84] E. Myers: Efficient Applicative Data Types, ACM Symp. on Principles of Programming Languages, 1984, 66-75.

[Wi87] D.S. Wise: Matrix Algebra and Applicative Programming, Conf. on Functional Programming and Computer Architectures, LNCS 274, 1987, 134-153.

A Algorithms for Fold

A.1 First Algorithm

Each process corresponding to a bag element in the fold algorithm executes the following (simplified) C-code:

```
#define UNDEF -1
#define n ???   /* size of bag */
typedef struct { int owner;  slock_t sem} Node;
shared Node tree[2*n-1]; shared T bag[n], *result; shared int weight[n];

void process(pid)
int pid;
{ int node, partner, my_leaf, partner_leaf;
my_leaf = pid mod n;
if (weight[my_leaf] == n) {result = &(bag[my_leaf]); exit;}
node = my_leaf;
while (node > 0)
  {node = (--node) / 2;
   S_LOCK(&(tree[node].sem));
   partner = tree[node].owner; partner_leaf = partner mod n;
   if (partner == UNDEF || tree[partner_leaf].owner != partner)
     {                         /* node has no owner; take possession of it */
       tree[node].owner = pid; S_UNLOCK(&(tree[node].sem));
       if (tree[my_leaf].owner != pid) break;}
   else
     { S_LOCK(&(tree[partner_leaf].sem));
       if (tree[partner_leaf].owner != partner)
         {                     /* owner of node has been combined; take node */
           tree[node].owner = pid;
           S_UNLOCK(&(tree[partner_leaf].sem)); S_UNLOCK(&(tree[node].sem));}
       else
         { S_LOCK(&(tree[my_leaf].sem));
           if (tree[my_leaf].owner != pid)
             {                 /* my bag element has already been combined; stop */
               S_UNLOCK(&(tree[my_leaf].sem)); S_UNLOCK(&(tree[partner_leaf].sem));
               S_UNLOCK(&(tree[node].sem)); break;}
            else               /* partner found; combine elements */
             { tree[partner_leaf].owner = UNDEF; tree[node].owner = UNDEF;
               S_UNLOCK(&(tree[my_leaf].sem)); S_UNLOCK(&(tree[partner_leaf].sem));
               S_UNLOCK(&(tree[node].sem));
               weight[my_leaf] += weight[partner_leaf];
               if (weight[my_leaf] == n) {result = &(bag[my_leaf]); break;}
               pid += n; node = my_leaf; create_process(my_leaf,partner_leaf);}}}
  } /* while */
exit; /* stop process */
} /* process */
```

`create_process(i1,i2)` computes `bag[i1]` $\oplus$ `bag[i2]` and delivers the result in `bag[i1]`. For simplicity, the new bag element inherits the current process.

A.2 Modified Fold Algorithm

```
typedef struct { int owner; slock_t sem } Node;
shared Node tree[2*n-1]; shared T bag[n], *result; shared int weight[n]; /* n>1 */
void process(pid)
int pid;
{ int partner, son, node;
node = pid; son = pid;
while (node > 0)
 { son = node; node = (--son) / 2;
   S_LOCK(&(tree[node].sem)); S_LOCK(&(tree[son].sem));
   if (tree[son].owner != pid)
    {                     /* the son has already been stolen, stop */
      S_UNLOCK(&(tree[son].sem)); S_UNLOCK(&(tree[node].sem)); break;}
   else
    {                     /* release the son */
      tree[son].owner = UNDEFINED; S_UNLOCK(&(tree[son].sem));
      if (tree[node].owner == UNDEFINED)
        {                 /* node is free; take possession of it */
          tree[node].owner = pid; S_UNLOCK(&(tree[node].sem));}
      else
        {                 /* partner found; combine elements */
          partner = tree[node].owner; tree[node].owner = UNDEFINED;
          S_UNLOCK(&(tree[node].sem)); weight[pid] += weight[partner];
          if (weight[pid] == n) {result = &(bag[pid]); break;}
          node = pid; create_process(pid,partner);}}
 } /* while */
exit;   /* stop process */
} /* process */
```